8-1995

Forcing Behavioral Subtyping Through Specification Inheritance

Krishna Kishore Dhara
Iowa State University

Gary T. Leavens
Iowa State University

Forcing Behavioral Subtyping Through Specification Inheritance

Abstract
A common change to object-oriented software is to add a new type of data that is a subtype of some existing type in the program. However, due to message passing unchanged parts of the program may now call operations of the new type. To avoid reverification of unchanged code, such operations should have specifications that are related to the specifications of the appropriate operations in their supertypes. This paper presents a specification technique that uses inheritance of specifications to force the appropriate behavior on the subtype objects. This technique is simple, requires little effort by the specifier, and avoids reverification of unchanged code. We present two notions of such behavioral subtyping, one of which is new. We show how to use these techniques to specify examples in C++.

Keywords
object-oriented programming, behavior subtype, abstract data type, mutation, simulation function, modular verification, supertype abstraction, interface specification, Larch/C++, class, C++, subclass

Disciplines
Programming Languages and Compilers

Forcing Behavioral Subtyping Through Specification Inheritance

Krishna Kishore Dhara and Gary T. Leavens

TR #95-20a
August 1995, revised August 1995

Keywords: object-oriented programming, behavioral subtype, abstract data type, mutation, simulation function, modular verification, supertype abstraction, interface specification, Larch/C++, class, C++, subclass.

1994 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented Programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory — semantics; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs — Abstract data types, modules, packages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Pre- and post-conditions; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Specification techniques.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Forcing Behavioral Subtyping Through Specification Inheritance

Krishna Kishore Dhara and Gary T. Leavens*
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
dhara@cs.iastate.edu and leavens@cs.iastate.edu

August 29, 1995

Abstract

A common change to object-oriented software is to add a new type of data that is a subtype of some existing type in the program. However, due to message passing unchanged parts of the program may now call operations of the new type. To avoid reverification of unchanged code such operations should have specifications that are related to the specifications of the appropriate operations in their supertypes. This paper presents a specification technique that uses inheritance of specifications to force the appropriate behavior on the subtype objects. This technique is simple, requires little effort by the specifier, and avoids reverification of unchanged code. We present two notions of such behavioral subtyping, one of which is new. We show how to use these techniques to specify examples in C++.

1 Introduction

Object-oriented (OO) software can be extended by adding new subtypes to existing types. Such extensions provide reuse of existing functions by allowing one to use subtype objects in place of supertype objects. However, due to message passing unchanged functions will then execute operations of

the newly added subtypes, potentially requiring respecification and reverification of the function [10]. Respecification and reverification go against the ease of extension promised by proponents of OO software. Specification and verification techniques which evolve with software, that is which do not require respecifying or reverifying whenever new components are added to the system, are called *modular*.

Behavioral subtyping, subtyping based on the behavior of types, can be used for modular specification and verification of OO software. A set of conditions for behavioral subtyping has been proposed both proof-theoretically [1, 12], and model-theoretically [10, 5]. It has been shown that with the addition of new behavioral subtypes, existing unchanged software does not have surprising behavior [10, 5]. Leavens and Weihl [11] present a technique for modular verification of OO programs. But to use such a technique one needs to verify that each specified subtype relation constitutes a behavioral subtype.

In this paper we present a modular specification technique, which automatically forces behavioral subtyping (and thus also avoids reverification). We also define a new, weaker notion of behavioral subtyping that permits more behavioral subtype relations than previous work [12]. Though we use C++ [17] as an OO language and Larch/C++ [4, 9] as a specification language, the techniques we present can also be applied to other programming languages and with other specification languages.

2 Background on Larch/C++

Larch [6] is a family of specification languages with a two-level approach to specification. One level of specification, the interface language, describes the interface and behavior of the modules of a programming language like C++ and Modula-3. Larch/C++ plays this role in this paper. The other component, the Larch Shared Language (LSL), describes the underlying model and the vocabulary that can be used in the interface language. For lack of space we sometimes omit details of the LSL traits.

Figures 1 and 2 give the specification of a C++ class, `BankAccount` in Larch/C++. The interface specification is given in Figure 1. The first `uses` clause in Figure 1 says that the abstract values of `BankAccount` objects and the vocabulary used to specify them are given by the trait `BankAccountTrait`

```
class BankAccount {
  public:
    uses BankAccountTrait;
    constraint self^.owner = self'.owner;
    BankAccount(long int cts, const char *name) { //constructor
       uses cpp_const_char_string;
       requires 0 < dollars(cts) ∧ nullTerminated(name, pre);
       modifies self;
       ensures  self' = [dollars(cts), uptoNull(name, pre)];
    }
    virtual long int balance() const {
       ensures approx_equal(dollars(result), self^.credit);
    }
    virtual void withdraw(long int cts) {
       requires 0≤dollars(cts) ∧ dollars(cts)≤self^.credit;
       modifies self;
       ensures  self' = set_credit(self^,
                                   self^.credit - dollars(cts));
    }
    // ... pay_interest and deposit are omitted
};
```

Figure 1: Interface Specification for `BankAccount`

```
BankAccountTrait: trait
  includes long, Rational(long for Int),
    String(char for E, String[char] for C),
    NoContainedObjects(BankAccount)
  BankAccount tuple of credit:Q, owner:String[char]
  introduces
    dollars: long → Q
    approx_equal: Q, Q → Bool
  asserts ∀ q1,q2:Q, c: long
    dollars(c) == c/100;
    approx_equal(q1, q2) == abs(q1 - q2) < (1/200)
```

Figure 2: Specification of `BankAccountTrait`

(in Figure 2). The abstract values are defined as tuples with a `credit` component and an `owner` component. The `constraint` clause in Figure 1 states that the `owner` of `BankAccount` cannot be changed. Following this history constraint [12] in Figure 1 are the specification of the C++ member functions.

The pre-condition for the constructor `BankAccount` follows the keyword `requires`. It requires that the amount be positive and that the name be a valid C++ string. (The trait function `nullTerminated` is specified in the trait listed by the `uses` clause in the constructor.) The `modifies` clause is a frame axiom. It states that only `self`, the receiver of the message, can be changed by the function. The postcondition, given by the `ensures` clause, forces the value of the `self` in the post-state to be the tuple with `cts` converted to dollars and the string denoted by `name` in the pre-state. Specification for the member function `balance` ensures that the result is approximately equal to the amount in `credit` converted into dollars. The member function `withdraw` modifies self by decreasing its credit component by `dollars(cts)`, provided `cts` is non-negative and is less than the credit of self in the pre-state.

After specifying and implementing `BankAccount`, a programmer might define functions to manipulate `BankAccount` objects. Figure 3 gives the Larch/C++ specification of such a function. The pre-condition prevents the

```
imports BankAccount;
void transfer(BankAccount& source,
              BankAccount& sink,
              long int cts) {
  let amt: Q be dollars(cts);
  requires source ¬= sink ∧ source^.credit ≥ amt
           ∧ amt ≥ 0;
  modifies source, sink;
  ensures  sink' = set_credit(sink^, sink^.credit + amt)
           ∧ source' = set_credit(source^, source^.credit - amt);
}
```

Figure 3: Specification of the function `transfer`

two accounts from being the same object and requires the amount being transferred to be non-negative. The post-condition describes the transfer from `source` to `sink`. The specification uses the trait `BankAccountTrait` for its model and vocabulary, by importing the interface for `BankAccount`.

3 The Problem

One set of problems caused by OO programming techniques is illustrated by the following scenario. Suppose that, after using the type `BankAccount`, a new type of account, `PlusAccount`, is added to the program. A `PlusAccount` object has both a savings and a checking account. `PlusAccount` is intended to be a behavioral subtype of `BankAccount`. Such an extension does not change the existing code either for `BankAccount` or for `transfer`.

3.1 Specification and verification problems

Subtype polymorphism allows one to send `PlusAccount` objects in place of `BankAccount` objects as arguments to `transfer`. However, Figure 3 specifies `transfer` using the vocabulary for `BankAccount` values defined

in `BankAccountTrait`. How can one interpret what the specification of `transfer` says about `PlusAccount` objects, which might have completely different abstract values? For example, what does the trait function `.credit` mean when applied to a `PlusAccount`? Do we need to respecify `transfer` or do we need to use the same abstract values and trait functions when `PlusAccount` is added? Respecification goes against the modularity of OO software. Using the same abstract values and trait functions would not work in this example, because `PlusAccount` contain more information than `BankAccount` objects. One alternative would be to specify all abstract values as tuples, and treat subtype values in supertype contexts by projecting away the extra components, as is done in other OO specification languages (such as Object-Z, VDM++, MooZ, and Z++ [8]). But this would prevent viewing `PlusAccount` objects as `BankAccount` objects by adding the two account balances together. A solution should be flexible, by allowing users to specify the relationship between the abstract values of subtypes and supertypes, but should be modular.

To see the verification problem, consider the case that `transfer` is verified, before `PlusAccount` is added, using the specification of `BankAccount`. When the subtype `PlusAccount` is added, the implementation of `transfer` is not changed, but the code it executes changes when `PlusAccount` objects are passed. In such cases, the code will execute methods with different specifications than those used during verification. Therefore the verification of `transfer` might no longer be valid. Reverifying `transfer` for such cases, using the new specifications of methods of `PlusAccount` would solve this problem. However, reverifying all unchanged functions whenever new subtypes are added is not practical or desirable. A modular verification technique is needed to avoid reverification.

3.2 Overview of the Solution

Our approach to solving the specification problem is to require that behavioral subtypes provide a way to interpret the mathematical vocabulary of their supertypes. In this paper the interpretation is given by specifying a simulation function, which is a mapping from the abstract values of a subtype to those of its supertypes. The use of such functions dates back to Hoare's work [7], and has been used in the context of subtyping by other authors [15, 2, 1, 12].

If all the subtypes used in a program are behavioral subtypes, the technique of supertype abstraction can be used for modular verification of OO programs [10]. Supertype abstraction uses static types of variables to reason about code and restricts the runtime types of variables to be behavioral subtypes of the static types. Such verification is valid because of the semantic conditions imposed on behavioral subtypes, which constitute an additional proof obligation. When new behavioral subtypes are added such a technique does not require reverification because subtype objects behave like supertype objects.

In this paper we illustrate how these two techniques are combined in Larch/C++ to give a semantics for specification inheritance that forces subtypes to be behavioral subtypes (following [18]). We also define a new, weaker notion of behavioral subtyping for mutable types, which has advantages over Liskov and Wing's definition [12].

In the next section we discuss different notions of behavioral subtyping. In Section 5 we describe semantics of specification inheritance and show how specification inheritance forces behavioral subtyping. Section 6 discusses our techniques, and the last section presents our conclusions.

4 Behavioral Subtyping

In contrast to structural subtyping [3], behavioral subtyping should be based on both the syntax and the semantics of types. That is, behavioral subtyping is a property that relates type specifications. Syntactic constraints, as in structural subtyping, ensure that an expression of a subtype can be used in place of an expression of its supertype without any type error. Semantic constraints ensure that subtype objects behave like supertype objects; that is, the use of subtype objects in place of supertype objects does not produce any unexpected behavior. Expected behavior is characterized by the supertype's specification.

To define behavioral subtyping, we use the following notation. We use *strong behavioral subtyping*, $\leq_{\mathrm{S}}$, to refer to a notion similar to Liskov and Wing's constraint-based behavioral subtyping [12] and *weak behavioral subtyping*, $\leq_{\mathrm{W}}$, to refer to our new, weaker notion of behavioral subtyping. Type symbols are represented by S, T and type vectors by $\vec{U}$, $\vec{V}$. A simulation function from type S to T is denoted by $c_{S \to T}$, an invariant of a type T,

by I_T, and a constraint of T, by C_T. The notation $pre^m_T(self, \vec{x})$ denotes the precondition of method m in T, with receiver $self$ and additional parameters $\vec{x}$. Substituting z for y in predicate $p(y)$ is written as $p(z)$. We use $x\backslash s$ for the value of x in state s, and $x\,\hat{}$ and x' for values of x in `pre` and `post` states respectively. (Hence $x\backslash$`pre` is equivalent to $x\,\hat{}$.)

We present the definitions of $\leq_S$ and $\leq_W$ in two steps. We first define pre-behavioral subtyping, which captures the common parts of these definitions. Then we define $\leq_S$ and $\leq_W$ using pre-behavioral subtyping, which highlights the differences between the two definitions. The definition of pre-behavioral subtyping uses ideas from [1, 12]. The definition is specialized for single dispatching languages like C++, Eiffel, and Smalltalk.

Definition 4.1 (Pre-Behavioral Subtyping) *S is a pre-behavioral subtype of T with respect to a binary relation $\leq$ on types if the following properties are satisfied.*
Syntactic *For every method m in T there exists a method m in S such that the following hold.*

- *Contravariance of arguments. If the types of the additional arguments of m in S and T are $\vec{U}$ and $\vec{V}$ respectively, then $|\vec{U}| = |\vec{V}| \wedge (\forall i. V_i \leq U_i)$.*
- *Covariance of result. If the result types of m in S and T are U_r and V_r respectively, then $U_r \leq V_r$.*
- *Exception rule. Any exception result in S is of a type that is related by $\leq$ to the corresponding exception type in T.*

Semantic *There exists a family of simulation functions, $\langle c_{U \to V} : U \leq V \rangle$, such that the following hold.*

- *Invariant rule. For all values v_S of S, $I_S(v_S) \Rightarrow I_T(c_{S \to T}(v_S))$*
- *Methods rule. For all common methods m, if the additional arguments types of m in S and T are $\vec{U}$ and $\vec{V}$ respectively and the result types of m in S and T are S_r and T_r respectively, then for all objects $self : S$, $\vec{y} : \vec{V}$, and $result : S_r$*
  - *Precondition rule.*

$$pre^m_T(c_{S \to T}(self\,\hat{}), \vec{y}\,\hat{}) \Rightarrow pre^m_S(self\,\hat{}, c_{\vec{V} \to \vec{U}}(\vec{y}\,\hat{})),$$

– *Postcondition rule.*

$$
\begin{aligned}
&(pre_S^m(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}))\\
&\qquad \Rightarrow post_S^m(self\hat{}, self', c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}), c_{\vec{V}\to\vec{U}}(\vec{y}'), result'))\\
&\Rightarrow (pre_T^m(c_{S\to T}(self\hat{}), \vec{y}\hat{})\\
&\qquad \Rightarrow post_T^m(c_{S\to T}(self\hat{}), c_{S\to T}(self'), \vec{y}\hat{}, \vec{y}', c_{U_r\to V_r}(result')))
\end{aligned}
$$

In C++ the additional arguments of a method must have the same types as in the corresponding method of the supertype, otherwise overloading instead of inheritance results. (Also, in C++ one can think of *self* as `*this`.) Therefore, for C++ we can state the methods rule more simply as follows (compare [12]).

> *For all common virtual functions m, if the additional arguments types of m in S and T are $\vec{V}$ and the result types of m in S and T are S_r and T_r respectively, then for all objects $self : S$, $\vec{y} : \vec{V}$, and $result : S_r$*
>
> - *Precondition rule.*
>
> $$pre_T^m(c_{S\to T}(self\hat{}), \vec{y}\hat{}) \Rightarrow pre_S^m(self\hat{}, \vec{y}\hat{}),$$
>
> - *Postcondition rule.*
>
> $$
> \begin{aligned}
> &(pre_S^m(self\hat{}, \vec{y}\hat{})\\
> &\qquad \Rightarrow post_S^m(self\hat{}, self', \vec{y}\hat{}, \vec{y}', result'))\\
> &\Rightarrow (pre_T^m(c_{S\to T}(self\hat{}), \vec{y}\hat{})\\
> &\qquad \Rightarrow post_T^m(c_{S\to T}(self\hat{}), c_{S\to T}(self'), \vec{y}\hat{}, \vec{y}', c_{U_r\to V_r}(result')))
> \end{aligned}
> $$

An additional semantic condition on the history constraints of the types distinguishes between strong and weak behavioral subtyping. History constraints are introduced by Liskov and Wing in order to capture the properties of objects that are true in any execution history (which Liskov and Wing call computation) [13]. For example, the history constraint of `BankAccount` specified in Figure 1 states that the name of the owner does not change in any computation.

4.1 Strong behavioral subtyping

The following definition is a modified version of Liskov and Wing's definition [12, Figure 4]. The exception rule and the methods rule are changed from the original definition (see the section on related work below for details on the differences).

Definition 4.2 (Strong Behavioral Subtyping, $\leq_S$) *$S \leq_S T$ if S is a pre-behavioral subtype of T with respect to $\leq_S$, and the following constraint rule is satisfied.*
Semantic

- *Constraint rule. For all valid computations, c, for all states pre and $post$ in c such that pre precedes $post$, and for all objects $self : S$,*

$$C_S(self\backslash pre, self\backslash post) \Rightarrow C_T(c_{S\to T}(self\backslash pre), c_{S\to T}(self\backslash post)).$$

Unlike Liskov and Wing's definition, the exception rule for $\leq_S$ allows subtype methods to return objects of a behavioral subtype of the supertype's exception type. This change is necessary for languages like C++ where exceptions are allowed to return subtype objects[1]. Since the semantic conditions on exceptions and exception results are captured in the postconditions, only the syntactic condition on exceptions is stated explicitly.

The syntactic conditions can be checked by the type system of a language. But the semantic conditions need to be verified and are generally beyond the power of most type systems to check. Several examples of $\leq_S$, like `Bag` $\leq_S$ `PriorityQueue`, are given by Liskov and Wing [12]. The `PlusAccount` referred in Section 2 is a strong behavioral subtype of `BankAccount`. A detailed discussion of that example is provided in the next section.

4.2 Weak behavioral subtyping

The constraint rule in the definition of strong behavioral subtyping requires even the extra mutators in subtypes to satisfy the history constraints of the supertype. The only kinds of mutations permitted by this requirement are

[1] Technically, in C++ one has to use a pointer or a reference.

those that are possible in the supertype or those that mutate extra state (state lost in simulation functions) in the subtype objects. For example, a type of immutable arrays cannot have any strong behavioral subtypes that allow changing elements.

However, when subtype objects are passed in place of supertypes, extra mutators in the subtypes are not visible, because subtype objects are viewed from supertype's methods. If appropriate restrictions are placed on certain forms of aliasing (discussed below) then one can allow extra mutations in the subtype and can still expect subtype objects to behave like supertype objects when viewed through the supertype's methods [5]. Since this notion allows more behavioral subtypes by weakening the constraint rule, it is called weak behavioral subtyping.

Definition 4.3 (Weak behavioral subtyping, $\leq_{\mathbf{W}}$) *$S \leq_w T$ if S is a pre-behavioral subtype of T with respect to $\leq_w$, and the following constraint rule is satisfied.*
Semantic

- *Constraint rule. For all valid computations, c, which do not invoke extra methods of S, for all states pre and post in c such that pre precedes post, and for all objects self : S,*

$$C_S(self\backslash pre, self\backslash post) \Rightarrow C_T(c_{S\rightarrow T}(self\backslash pre), c_{S\rightarrow T}(self\backslash post)).$$

Another way of interpreting weak behavioral subtyping is to view the supertype's history constraint as part of the postcondition of each of its methods. In such a case, when weak behavioral subtypes are specified, the postconditions of the extra methods in the subtypes need not include the supertype's history constraint.

The main drawback of weak behavioral subtyping is that a restriction on certain kinds of aliasing is required for modular verification. Since the supertype's constraints might not be satisfied by the extra mutators of the subtype, manipulating an object simultaneously both from a supertype's point of view and from a subtype's point of view will result in a strange behavior. One must restrict direct aliasing between variables or objects of different types. For a detailed discussion on techniques to restrict such forms of aliasing and for a model theoretic definition of weak behavioral subtyping

see [5]. Other forms of aliasing such as direct aliasing between variables and objects of the same type and indirect aliasing (that is aliasing of components) between variables and objects of different types is allowed.

Weak behavioral subtyping captures several useful behavioral subtype relationships not captured by strong behavioral subtyping, including mutable types that are weak behavioral subtypes of immutable types. One such example is that a mutable array type can be a subtype of an immutable array type, which allows one to pass a mutable array as an argument to a function, like finding the maximum array element, that expects an immutable array. Similarly, a mutable record type can be specified as a weak behavioral subtype of an immutable record type with fewer fields. One can have a hierarchy of weak behavioral subtypes with varying degrees of mutability [5].

5 Specification Inheritance

To prove a strong or a weak behavioral subtyping relation between two types, one needs to prove that the conditions of the appropriate definitions are met. In this section we show how specification inheritance can be used to force behavioral subtyping, eliminating the need for users of a specification language to verify behavioral subtyping by hand[2]. This idea is due to Wills [18] although apparently he allows one to escape the mechanism and still specify subtypes that are not behavioral subtypes.

5.1 Inheritance for strong behavioral subtyping

The specification of `PlusAccount` in Figures 4 and 5 gives an example of specification inheritance. Figure 4 gives the interface specification. From the `uses` clause and the trait in Figure 5 one can see that the abstract values of `PlusAccount` objects are tuples with savings, checking and owner components. The `simulates` clause states that `PlusAccount` is a strong behavioral subtype of `BankAccount` (because it is not written `weakly simulates`). It also says that the simulation function `toBA` is used to view `PlusAccount`

[2]If the specification given in the subtype contradicts a supertypes' specification then the type may not be implementable, so some verification of implementability might still be useful.

```
imports BankAccount;
class PlusAccount : public BankAccount {
  //PlusAccount subtype of BankAccount
  uses PlusAccountTrait;
  simulates BankAccount by toBA;
  public:
    // constructor, pay_interest, deposit, and
    // credit_check are omitted
    // balance specification is inherited
    virtual void withdraw(long int cts) {
      let amt:Q be dollars(cts);
      requires 0 ≤ amt ∧ amt ≤ self^.savings;
      modifies self;
      ensures  self'.checking = self^.checking;
    }
};
```

Figure 4: Interface Specification of `PlusAccount`

values as `BankAccount` values. The simulation function `toBA` is defined in `PlusAccountTrait` in Figure 5.

The specification in Figure 4 inherits its invariant, history constraint, and parts of method specifications from the specification in Figure 1. For example, the specification of `withdraw` is partly inherited and partly given in Figure 4. The specification of `withdraw` in Figure 4 only states that when the savings part is greater than the amount, then the checking part does not change. The inherited part of the specification states that the amount, if less than the `PlusAccount`'s credit, is deducted from the `PlusAccount`'s credit.

As an aid to explaining the semantics of specification inheritance, consider the completed specification of `withdraw` method. Its precondition is formed as a disjunction of the added precondition in the subtype with the supertype's precondition, after coercing subtype values to supertype values. That is, the completed precondition for `withdraw` is:

```
requires (0 ≤ dollars(cts)
```

```
PlusAccountTrait: trait
  includes BankAccountTrait,
           NoContainedObjects(PlusAccount)
  PlusAccount tuple of savings:Q, checking:Q,
                       owner:String[char]
  introduces
    toBA: PlusAccount → BankAccount
  asserts ∀ q1,q2: Q, o: String[char]
    toBA([q1, q2, o]) == [q1+q2, o]
```

Figure 5: Trait for `PlusAccountTrait`

```
              ∧ dollars(cts) ≤ toBA(self^).credit)
      ∨ (0 ≤ amt ∧ amt ≤ self^.savings);
```

The first disjunct is inherited from `BankAccount`, and the second disjunct is the added precondition in `PlusAccount`. The modifies clause of the subtype's method lists all the objects in the supertype's and the subtype's modifies clauses. The completed modifies clause of `withdraw` is:

```
modifies self;
```

Its postcondition is formed as a conjunction of two implications. The first is that the subtype's added precondition implies the subtype's added postcondition. The second is that, after coercion, the supertype's precondition implies the supertype's precondition. That is, the completed postcondition for `withdraw` is:

```
ensures (0 < dollars(cts)
              ∧ dollars(cts) ≤ toBA(self^).credit)
           ⇒ (toBA(self') = set_credit(toBA(self^),
                       toBA(self^).credit - dollars(cts)))
     ∧ (0 ≤ dollars(cts) ∧ amt ≤ self^.savings)
           ⇒ (self'.checking = self^.checking);
```

In general, the modifies clause of the completed specification is larger than the modifies clause of the specification being inherited. In such cases

the consequent of each implication in the completed specification's postcondition lists the objects that are not in the corresponding modifies clause as unchanged. (See [9] for details.)

The completed specification's invariant is formed as a conjunction of supertype's invariant with appropriate coercions and subtype's invariant. The completed specification's history constraint is a conjunction of the supertype's constraint with appropriate coercions and the added constraint on the subtype. For example, the completed constraint for `PlusAccount` is:

```
constraint toBA(self^).owner = toBA(self').owner;
```

The following definition generalizes these notions for multiple supertypes. The notation used below is as follows. The set of all supertypes of a types S is given by $Sups(S)$ and the set of all methods of a type T is given by $meths(T)$. The predicates $added\_I_S(v)$, $added\_pre_S^m$, and $added\_post_S^m$ are the added predicates for invariant of S and the pre- and postconditions of a method m in S respectively.

Definition 5.1 (Specification inheritance) *Let S be specified as a subtype of all the types in $Sups(S)$. Let $\langle c_{U\to V} : U \leq V \rangle$ be the specified family of simulation functions. The completed specification of S is:*

Invariant *$I_S(v)$ is:*

$$added\_I_S(v) \wedge \left(\bigwedge_{T \in Sups(S)} I_T(c_{S\to T}(v)) \right).$$

Precondition *for all methods m of S, $pre_S^m(self\hat{}, \vec{x}\hat{})$ is:*

$$added\_pre_S^m(self\hat{}, \vec{x}\hat{})$$

$$\vee \left(\bigvee_{\substack{T \in Sups(S),\\ m \in meths(T)}} \begin{array}{l} (\exists \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = \vec{x}\hat{} \\ \quad \Rightarrow pre_T^m(c_{S\to T}(self\hat{}), \vec{z}\hat{})) \end{array} \right).$$

Postcondition *for all methods m of S, $post^m_S(self\hat{}, self', \vec{x}\hat{}, \vec{x}', result')$ is:*

$$(added\_pre^m_S(self\hat{}, \vec{x}\hat{}) \Rightarrow added\_post^m_S(self\hat{}, self', \vec{x}\hat{}, \vec{x}', result'))$$

$$\wedge \left(\bigwedge_{\substack{T \in Sups(S),\\ m \in meths(T)}} \begin{array}{l} (\forall \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = \vec{x}\hat{} \wedge c_{\vec{V}\to\vec{U}}(\vec{z}') = \vec{x}' \\ \quad \Rightarrow (pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{}) \\ \qquad \Rightarrow (post^m_T(c_{S\to T}(self\hat{}), c_{S\to T}(self'), \\ \qquad\qquad \vec{z}\hat{}, \vec{z}', c_{U_r\to V_r}(result'))))) \end{array} \right).$$

Constraint *for all states pre, post, C_S is: for all objects $self : S$*

$$added\_C_S(self\backslash pre, self\backslash post)$$

$$\wedge \left(\bigwedge_{T \in Sups(S)} C_T(c_{S\to T}(self\backslash pre), c_{S\to T}(self\backslash post)) \right).$$

We can simplify the precondition and the postcondition rules for C++, because additional arguments to inherited methods cannot have different types. The precondition rule for specification inheritance in Larch/C++ is given below.

> *for all common virtual member functions m of S, $pre^m_S(self\hat{}, \vec{x}\hat{})$ is:*
>
> $$added\_pre^m_S(self\hat{}, \vec{x}\hat{}) \vee \left(\bigvee_{\substack{T \in Sups(S),\\ m \in meths(T)}} pre^m_T(c_{S\to T}(self\hat{}), \vec{x}\hat{}) \right).$$

The postcondition rule is:

> *for all common virtual member functions m of S, $post^m_S(self\hat{}, self', \vec{x}\hat{}, \vec{x}', result')$ is:*
>
> $(added\_pre^m_S(self\hat{}, \vec{x}\hat{}) \Rightarrow added\_post^m_S(self\hat{}, self', \vec{x}\hat{}, \vec{x}', result'))$

$$\wedge \left(\begin{array}{ll} & pre_T^m(c_{S\rightarrow T}(self\hat{}), \vec{x}\,\hat{}) \\ \bigwedge & \Rightarrow (post_T^m(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'), \\ T \in Sups(S), & \qquad \vec{x}\,\hat{}, \vec{x}\,', c_{U_r\rightarrow V_r}(result'))) \\ m \in meths(T) & \end{array} \right).$$

Since message passing in C++[17] is dynamic for only the virtual member functions, the semantics of specification inheritance in Larch/C++ applies the above rules for only the virtual member functions of S. As an example, the inherited specification of `balance` in `PlusAccount` is as follows:

```
virtual long int balance() const {
  ensures approx_equal(dollars(results),
                       toBA(self^).credit);
}
```

For some methods, such as `withdraw` the completed specification obtained by using specification inheritance is difficult to understand (see above). Separating the inherited part from the added part, we believe, enchances the readability of completed specifications and serves as an aid to understanding. This separation can be achieved by using case-analysis [18]. Case-analysis is a syntactic sugar used in method specifications. For example, Figure 6 gives the completed specification of `withdraw` in `PlusAccount` using case-analysis. The body of the specification contains two parts. The first is the added specification and the second is the inherited specification. The semantics is that an implementation must satisfy both parts. Therefore, the meaning is the same as before. However, this form of specification clearly shows that behavioral subtypes must satisfy their supertypes' specifications. Thus, we believe this form of specification would be useful in specification browsers.

The following theorem ties specification inheritance to strong behavioral subtyping.

Theorem 5.2 *Each type specified as a subtype using specification inheritance is a strong behavioral subtype of its supertypes.*

Proof: Suppose S is specified as a subtype of T. Since the syntactic hold because S is a subtype of T, we check the semantic conditions as follows.

```
virtual void withdraw(long int cts) {
  let amt:Q be dollars(cts);
  requires 0 ≤ amt ∧ amt ≤ self^.savings;
  modifies self;
  ensures self'.checking = self^.checking;

  ensures 0 ≤ dollars(cts)
          ∧ dollars(cts) ≤ toBA(self^).credit;
  modifies self;
  requires self' = set_credit(toBA(self^),
                       toBA(self^).credit - dollars(cts));
}
```

Figure 6: Completed specification of `withdraw` in `PlusAccount`

Invariant For all $v : S$ we calculate as follows.

$$
\begin{array}{ll}
 & I_S(v) \\
= & \{\text{by specification inheritance and definition of invariant}\} \\
 & added\_I_S(v) \wedge I_T(c_{S\to T}(v)) \\
\Rightarrow & \{\text{by } A \wedge B \Rightarrow B\} \\
 & I_T(c_{S\to T}(v))
\end{array}
$$

Constraint For all valid computations c, for all states pre and $post$ in c such that pre precedes $post$, and for all objects $self : S$,

$$
\begin{array}{ll}
 & C_S(self\backslash pre, self\backslash post) \\
= & \{\text{by definition of specification inheritance}\} \\
 & added\_C_S(self\backslash pre, self\backslash post) \\
 & \wedge\ C_T(c_{S\to T}(self\backslash pre), c_{S\to T}(self\backslash post)) \\
\Rightarrow & \{\text{by } A \wedge B \Rightarrow B\} \\
 & C_T(c_{S\to T}(self\backslash pre), c_{S\to T}(self\backslash post))
\end{array}
$$

Methods The following calculation uses the precondition rule of specifiction inheritance to show the precondition rule of behavioral subtyping. Let m be a common method in S and T, for all objects $self : S$, and $\vec{y} : \vec{V}$, and $result : S_r$.

$$
\begin{array}{ll}
 & pre^m_T(c_{S\to T}(self\hat{}), \vec{y}\hat{}) \\
= & \{\text{by } A \equiv (true \Rightarrow A)\} \\
 & c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \Rightarrow pre^m_T(c_{S\to T}(self\hat{}), \vec{y}\hat{}) \\
\Rightarrow & \{\text{by } \exists\text{-introduction}\} \\
 & (\exists \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \Rightarrow pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{})) \\
\Rightarrow & \{\text{by } B \Rightarrow (A \vee B)\} \\
 & added\_pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{})) \\
 & \quad \vee (\exists \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \Rightarrow pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{})) \\
\Rightarrow & \{\text{by definition of specification inheritance}\} \\
 & pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}))
\end{array}
$$

The following calculation uses postcondition rule of specification inheritance to complete the proof of methods rule. Let m be a common method in S and T, for all objects $self : S$ and, $\vec{y} : \vec{V}$, and $result : S_r$.

$$
\begin{array}{ll}
 & pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{})) \\
 & \quad \Rightarrow post^m_S(self\hat{}, self', c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}), c_{\vec{V}\to\vec{U}}(\vec{y}'), result') \\
= & \{\text{by definition of specification inheritance}\} \\
 & (added\_pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{})) \\
 & \quad \vee (\exists \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \Rightarrow pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{}))) \\
 & \Rightarrow (added\_pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{})) \\
 & \quad \Rightarrow added\_post^m_S(self\hat{}, self', c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}), c_{\vec{V}\to\vec{U}}(\vec{y}'), result')) \\
 & \quad \wedge (\forall \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \wedge c_{\vec{V}\to\vec{U}}(\vec{z}') = c_{\vec{V}\to\vec{U}}(\vec{y}') \\
 & \quad \Rightarrow (pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{}) \\
 & \quad \Rightarrow (post^m_T(c_{S\to T}(self\hat{}), c_{S\to T}(self'), \\
 & \quad \vec{z}\hat{}, \vec{z}', c_{U_r\to V_r}(result')))))) \\
\Rightarrow & \{\text{by } A \wedge B \Rightarrow B\} \\
 & (added\_pre^m_S(self\hat{}, c_{\vec{V}\to\vec{U}}(\vec{y}\hat{})) \\
 & \quad \vee (\exists \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \Rightarrow pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{}))) \\
 & \Rightarrow (\forall \vec{z} : \vec{V}.c_{\vec{V}\to\vec{U}}(\vec{z}\hat{}) = c_{\vec{V}\to\vec{U}}(\vec{y}\hat{}) \wedge c_{\vec{V}\to\vec{U}}(\vec{z}') = c_{\vec{V}\to\vec{U}}(\vec{y}') \\
 & \quad \Rightarrow (pre^m_T(c_{S\to T}(self\hat{}), \vec{z}\hat{}) \\
 & \quad \Rightarrow (post^m_T(c_{S\to T}(self\hat{}), c_{S\to T}(self'), \\
 & \quad \vec{z}\hat{}, \vec{z}', c_{U_r\to V_r}(result')))))
\end{array}
$$

$\Rightarrow$ {by $(A \vee B \Rightarrow C) \Rightarrow (B \Rightarrow C)$}
$(\exists \vec{z} : \vec{V}.c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \Rightarrow pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{z}\,\hat{}))$
$\Rightarrow (\forall \vec{z} : \vec{V}.c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \wedge c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,') = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,')$
$\Rightarrow (pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{z}\,\hat{})$
$\Rightarrow (post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'),$
$\vec{z}\,\hat{}, \vec{z}\,', c_{U_r\rightarrow V_r}(result')))))$

$\Rightarrow$ {by $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$, where A is the first line below, B is the first line in the formula above, and $A \Rightarrow B$ by $\exists$-I}
$(c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \Rightarrow pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{}))$
$\Rightarrow (\forall \vec{z} : \vec{V}.c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \wedge c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,') = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,')$
$\Rightarrow (pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{z}\,\hat{})$
$\Rightarrow (post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'),$
$\vec{z}\,\hat{}, \vec{z}\,', c_{U_r\rightarrow V_r}(result')))))$

$=$ {by $(true \Rightarrow A) \equiv A$}
$pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow (\forall \vec{z} : \vec{V}.c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \wedge c_{\vec{V}\rightarrow\vec{U}}(\vec{z}\,') = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,')$
$\Rightarrow (pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{z}\,\hat{})$
$\Rightarrow (post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'),$
$\vec{z}\,\hat{}, \vec{z}\,', c_{U_r\rightarrow V_r}(result')))))$

$=$ {by instantiation}
$pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow (c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,\hat{}) \wedge c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,') = c_{\vec{V}\rightarrow\vec{U}}(\vec{y}\,'))$
$\Rightarrow (pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow (post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'),$
$\vec{y}\,\hat{}, \vec{y}\,', c_{U_r\rightarrow V_r}(result')))))$

$=$ {by $(true \Rightarrow A) \equiv A$}
$pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow (pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow (post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'),$
$\vec{y}\,\hat{}, \vec{y}\,', c_{U_r\rightarrow V_r}(result')))))$

$=$ {by $A \Rightarrow (A \Rightarrow B) \equiv (A \Rightarrow B)$}
$(pre^m_T(c_{S\rightarrow T}(self\hat{}), \vec{y}\,\hat{})$
$\Rightarrow post^m_T(c_{S\rightarrow T}(self\hat{}), c_{S\rightarrow T}(self'), \vec{y}\,\hat{}, \vec{y}\,', c_{U_r\rightarrow V_r}(result')))$

∎

The significance of this theorem is that strong behavioral subtyping is automatic for types specified using specification inheritance. From this the-

```
imports BankAccount;
class MutableAccount : public BankAccount {
  uses BankAccountTrait(MutableAccount for BankAccount);
  weakly simulates BankAccount by toBAwithoutChange;
  public:
    // constructor omitted
    // balance and withdraw are inherited
    virtual void change_name(const char *name) {
       uses cpp_const_char_string;
       requires nullTerminated(name, pre);
       modifies self;
       ensures self' = [self^.credit, uptoNull(name, pre)];
    }
};
```

Figure 7: `MutableAccount` as a weak behavioral subtype of `BankAccount`

orem we can conclude that `PlusAccount` is a strong behavioral subtype of `BankAccount`.

5.2 Inheritance for weak behavioral subtyping

Figure 7 gives the specification (using specification inheritance) of a type `MutableAccount`. The C++ syntax `:public BankAccount` in the declaration of `MutableAccount` states that it is a subtype of `BankAccount`. The `weakly simulates` clause states that `MutableAccount` is intended to be a weak behavioral subtype of `BankAccount`. If we use the specification inheritance rules discussed above we would inherit the history constraint of `BankAccount`, and would apply it for all the methods of `MutableAccount`. Since `change_owner` violates this history constraint, a different rule is needed for inheritance of history constraints to make weak behavioral subtypes.

For weak behavioral subtypes the inherited history constraint is applied only to the common methods. This condition allows the extra methods in the subtype, such as `change_name`, to mutate the state in a way that is not pos-

sible in the supertype. For `MutableAccount` the inherited constraint is given below. (The syntax "`for ....`" indicates to which methods the constraint applies.)

```
constraint identity(self^).name = identity(self').name
        for virtual long int balance(),
            virtual void withdraw(long int cts),
            virtual void pay_interest(double rate),
            virtual deposit(long int cts);
```

The proof of the following theorem is essentially the same as for the previous theorem.

Theorem 5.3 *Each type specified as a weak behavioral subtype using specification inheritance is a weak behavioral subtype of its supertypes.* ∎

6 Discussion

In this section we compare our work on behavioral subtyping and specification with other related work and also discuss issues in specification inheritance.

6.1 Related work

The important difference between our work and Liskov and Wing's work [12] is the new definition of weak behavioral subtyping. However we also refined their definition of strong behavioral subtyping. These refinements include changes to the exception rule, handling additional arguments in the methods rule, and generalizing the post-condition rule. The change in the exception rule is necessary to handle the case when the subtype objects are passed as exception results. But the change in the post-condition rule allows subtype methods to operate outside the domain of the supertype methods. For example consider the specification of a method given in both a supertype and a subtype.

```
virtual int foo(int x) { //supertype's specification
   requires x > 0;
  ensures result > 0;
}
```

```
virtual int foo(int x) { //subtype's specification
  requires x ≤ 0;
  ensures result = -3;
}
```

By our semantics of specification inheritance, the completed specification has the following postcondition.

```
((x > 0) ⇒ (result > 0) ∧ ((x ≤ 0) (result = -3))
```

This does not imply the supertype's postcondition. However, when one reasons about an invocation of `foo` on an object whose static type is the supertype, the subtype's `foo` performs adequately (without surprises). Hence their original rule is needlessly strong. Our definition of strong behavioral subtyping permits more strong behavioral subtype relationships, and gives the specifier more flexibility. Therefore although such a method specification could be used in a strong behavioral subtype according to our definition, it would not yield a strong behavioral subtype according to Liskov and Wing's original definition.

In Eiffel [14, Section 10.15], one can also inherit pre- and postconditions. Unlike the specification inheritance described in this paper, one can separately inherit the pre- and postconditions. (Eiffel's assertion sublanguage has no support for frame axioms like the modifies clause in Larch/C++.) The keyword "`else`" can be used at the start of a precondition to make the completed precondition be the disjunction of the supertype's precondition and the one stated. Similarly the keyword "`then`" can be used in the postcondition to make the completed postcondition be the conjunction of the supertype's postcondition and the one stated. Therefore, one can use specification inheritance in Eiffel to make subtypes that are not behavioral subtypes. In Eiffel there is no need for simulation functions in inherited specifications, as the assertion sublanguage is polymorphic by virtue of using Eiffel subexpressions. Eiffel's syntax provides no support for case-analysis in method specifications.

The work of Wills in Fresco [18] is most closely related to ours. Capsules in Fresco support the idea of case-analysis – all the specification capsules for a given method must be satisfied by that method. Wills has no way to write his "retrieval relations" into specifications of subtypes, however, making it difficult to apply the specification of supertypes to subtypes unless the

subtype has the same instance variable. Wills also does not force behavioral subtyping, as he allows users to escape from specification inheritance if desired.

6.2 Specification and verification

To solve the specification problem for the `transfer` function discussed in Section 3, one can use a technique similar to the one used in specification inheritance. Whenever a subtype of `BankAccount` is passed as an argument to `transfer`, the object's abstract value is coerced to a `BankAccount` abstract value using a simulation function. The vocabulary of `BankAccount` is then used to interpret the specification. For example, when `PlusAccount` object is passed to `transfer` one would coerce its value using `toBA`. However, there remains a problem of information loss with this technique. That is, the specification of `transfer` does not say how the amount transferred is distributed between checking and savings.

Since `PlusAccount` is a strong behavioral subtype of `BankAccount`, all the properties that are true for `BankAccount` objects are true for `PlusAccount` objects. The verification of `transfer` (done before `PlusAccount` was added) is valid even for `PlusAccount` objects passed to `transfer`. Reverification is not required.

However, in the case of weak behavioral subtypes, all the properties of the supertype are satisfied by weak behavioral subtype objects only when viewed as a supertype object. To avoid reverification, the programming method or verification logic should prevent aliases that allow a weak behavioral subtype object to be viewed both as a subtype and as a supertype. If that is done, then reverification for `MutableAccount` arguments is not needed.

A problematic feature of OO programming is the pervasiveness of objects. One can have abstract values that contain objects. Therefore, in general, simulation functions on values alone are not sufficient for an OO setting. One needs, at least, to give simulation functions access to the state functions, which map objects to their values (and so model a computer's memory). As an example, consider refined specifications of `PlusAccount` and `BankAccount` where one uses variables to document a design decision. Since the variable can be mutated, the abstract values contain objects that model them. Such abstract values for `BankAccount` and `PlusAccount` might look like the following (where Obj[int] represents an integer variable).

```
BankAccount tuple of dollars: Obj[int], cents: Obj[int],
                     owner: String[char]
PlusAccount tuple of svgs_dlrs: Obj[int], svgs_cents: Obj[int],
                     chkg_dlrs: Obj[int], chkg_cents: Obj[int],
                     owner: String[char]
```

A simulation function from the abstract values of `BankAccount` to the abstract values of `PlusAccount` is not enough, because a state is required to get the values inside objects such as `svgs_dlrs`. Thus one needs simulation functions to map states with subtype values to states with supertype values. Such a function should preserve aliasing. However, given the incremental nature of OO software development, specifying simulation functions from states to states is not practical. We need techniques to construct such simulation functions from coercions on individual types.

In this paper we have used simulation functions, which are convenient in formulas. However, in general, one needs relations instead of functions [16].

7 Conclusions

The main contributions of this paper are a modular specification technique which forces behavioral subtyping and a new, weaker notion of behavioral subtyping. While the semantics of behavioral subtyping may seem somewhat intricate, the basic idea is that the subtype must satisfy the supertype's specifications. This is enforced by our semantics of specification inheritance, and made visible by the case-analysis form of the completed specification.

References

[1] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, N.Y., 1991.

[2] Kim B. Bruce and Peter Wegner. An algebraic model of subtypes in object-oriented languages (draft). *ACM SIGPLAN Notices*, 21(10), October 1986.

[3] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, N.Y., 1991.

[4] Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[5] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference, Preliminary Proceedings*, pages 269–290. Elsevier, March 1995. To appear in *Electronic Notes in Computer Science*, volume 1.

[6] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[7] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[8] K. Lano and H. Haughton. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[9] Gary T. Leavens. Larch/C++ Reference Manual. Available in ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz., 1995.

[10] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[11] Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28d, Department of Computer Science, Iowa State

University, Ames, Iowa, 50011, August 1994. Full version of a paper to appear in *Acta Informatica*. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[12] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[13] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[14] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[15] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Proc. IFIP Congress '83, Paris*, September 1983.

[16] Oliver Schoett. An observational subset of first-order logic cannot specify the behavior of a counter. In C. Choffrut and M. Jantzen, editors, *STACS 91 8th Annual Symposium on Theoretical Aspects of Computer Science Hamburg, Germany, February 1991 Proceedings*, volume 480 of *Lecture Notes in Computer Science*, pages 499–510. Springer-Verlag, New York, N.Y., 1991.

[17] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., Reading, Mass., 1986. Corrected reprinting, 1987.

[18] Alan Wills. Specification in Fresco. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, chapter 11, pages 127–135. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.